# PNE Automation

*This is an outline of a case study presentation of PNE Automation. It is still a rough draft that needs a lot of distillation.*

*It is meant to be modular so that we can mix and match sections depending on the audience and context of the presentation. We should structure the presentation to work with and without code walkthroughs or an active internet connection.*

## Introduction

*This may or may not be more for internal talks and is meant to introduce the team and the project.*

* PNE Automation Team
  + Jim English – Development Manager
  + Charles Bryant – Automation Lead
  + Ryan White – Automation Developer
  + Mike Confer – Technical Lead
  + Michael Snow – Senior Developer
* Company/Product
  + SunGard
  + PayNetExchange PNE
    - Payment File Processing
* Milestones
  + Jim sets goals and direction for PNE Automation (2013)
  + Ryan began coding tests as a part of our grad program (July 2013)
  + Ryan finishes base test suite for Vendor Site (August 2013)
  + Ryan finishes base test suite for Customer Site (December 2013)
  + Charles moves from PNE production development to lead the PNE Automation project in the newly formed Automation Engineer role (December 2014)
* Purpose of Presentation
  + We want to share our experience with the community.
  + We want to get feedback so we can improve.

## Our Environment

*This needs to just give enough context to show why we made certain decisions.*

Our environment had an impact on many of the decisions that were made in our automation project.

* Web Applications with Web Forms
* Services with WCF and ASMX
* Thin Client with Basic JavaScript
  + In the current PNE version we don’t have to worry too much about JS in test. Basically JS provides validation and some minor functionality.
  + Show jQuery code
* C# Thick Codebehinds (sounds nasty and it’s even harder to unit test)
  + Show large god method in codebehind with line number in the hundreds, obvious null ref issues, deeply nested logic, all in on method
* SOA
  + Large API surface makes it impractical to test every method and permutation of service calls and application state.
  + Show service interface collapsed with method after method after method
* SQL Server
  + Good amount of logic in stored procedures and functions
  + No real segregation of read and write operations which leads to very little optimization of reads, but also means a lot of complexity going from a relational model to object oriented model. This has an impact on what and how we test.
  + No DBA
  + Show sample SQL with if statements, case statement, crazy joins and unions…
* Server Environment
  + ASP.Net hosted in IIS 7
  + Source Control
    - PNE - SVN
    - PNE Automation - Git
  + Dedicated Build Server (CruiseControl)
  + Dedicated Test Server
  + 2 Development Environments
  + QA Environment
  + Product Readiness/Staging Environment
  + Demo Environment
  + Production
* We don’t have a dedicated BA
  + This is a major gap in establishing a shared understanding of features

## Why Functional Testing

*Just an overview of the kind of testing we are focusing on and why we chose it based on the context in the previous section.*

* Functional Browser Based tests was the best solution to get test coverage that could sense functionality from UI down to SQL Server and across our APIs and Thick Codebehinds without having to do major refactorings to enable unit testing or complex integration tests.
* With functional tests in place it makes it more comfortable to refactor without the paranoia and change paralysis found in large legacy systems.

## Test Tools

*This is the beginning of the meat where we give an overview of the tools and there usage from a high level perspective.*

* NUnit/MSTest
  + Provides the core engine behind our functional tests. We use NUnit because it is the direction Ryan started with and some people on the team have an affinity to it and it provided the best support for reporting in CruiseControl. I added MSTest because I like to drink the Microsoft Kool-Aide and it provides the best integration with Microsoft Products. I am working on a test reporting engine that can relieve our dependence only on NUnit/CC.
* Selenium Web Driver
  + Intro
  + Element Location
  + Element Interaction
  + Navigating
  + Waits (Implicit and Explicit)
* SpecFlow
  + Intro
  + Gherkin/Cucumber
  + BDD
  + Book

## Test Framework

*This gets into more specifics of how we actually integrate the tools into a framework to run our tests.*

* + SpecFlow Feature
  + SpecFlow Scenarios
    - Defines test reusable test steps
  + SpecFlow Steps
  + Base Classes
  + Page Model
  + Test Configuration
    - Test Definition
    - Test Data
  + Sample Test (Login)
  + Current Plans to Expand the Framework
    - We intend to distill our current architecture to abstract additional objects to represent
      * Application
      * Application Environment
      * Browser
      * Page
      * Test Runner
      * Test Scenario
      * Test Feature
      * Test Step
      * Test Result (Core object in reporting engine)
    - Single Machine Multicore Parallel Testing
      * NUnit limitations
    - Distributed Testing (Remote or Virtual test nodes)
      * Selenium Grid

## Test Development Workflow

*This gives an overview of how test development fits in the overall SDLC.*

* It all starts with a ticket
* We have 3 types of tickets that may require test automation
  + Feature – new functionality
  + Defect – broken functionality in an existing feature
  + Incident – broken functionality in an existing feature that requires immediate attention
* Feature
  1. Product Management defines new functionality
  2. Automation Engineer writes spec that proves the functionality works and attaches to ticket
  3. Developer develops the functionality
  4. Automation Engineer/Developer writes test steps to implement the test spec
  5. The test is ran and any failures would require a loop back to #3.
* Defect/Incident
  1. QA writes a test that proves the functionality doesn’t work
  2. Automation Engineer writes spec that proves the functionality is fixed
  3. Go to Feature #3.

## Defining Specss

*Overview of how we decided on what to test first when the project started and how we decide what to test going forward.*

* What test to write first?
  + Starting out Ryan made truth tables to try to get adequate coverage of variations in arguments and state for various forms. Ryan would be writing tests the rest of his life and the next if he had to write truth table tests against some of our service layer.
  + Establish benchmark then push the bar up and to the right.
* SpecFlow Feature
  + Agreement with Biz and QA (This is what we are going to do, OK?)
  + Without dedicated BA it is difficult to have everything formally spec’d
  + QA doesn’t do upfront test plans that are published to the project team
  + When we write feature files we are distilling our understanding into a form that can be understood by QA and the Biz. They may never read it, but we should use it as the basis for Sprint Demos, walkthroughs, prototyping, and them to tickets. The more we make them visible the more they will become a part of the culture.

### Spec Tags

Spec Tags (@*{Name}*) are used to organize tests and enable isolation in test runs. The tags are actually SpecFlow constructs used to control test execution. If you look at the spec below, the tags allow us to categorize the test based on various attributes.

***@Req.4.4.2.7 @JMS @Recipients @RecipientDetails @Toolbar @Manual @OTF11533***

Scenario: Toolbar color, verify background is correct color.

When I open the Recipient Details page

Then the toolbar background color should be "#CCCCCC"

* @Req.4.4.2.7 – this tag signifies the requirement that the spec is based on. This provides traceability of the spec back to the requirement as long as the numbering doesn’t change in the requirement document.
* @JMS @Recipients @RecipientDetails @Toolbar – this tag is like the namespace for the unit of the application the spec defines:
  + @JMS – the application
  + @Recipient – the section of the application
  + @RecipientDetails – the page or control in the application
  + @Toolbar – the sub-page/control in the parent page/control. These tags should not be deeply nested. If nesting occurs, the spec should be broken out into a separate spec with a sub becoming a parent. This helps readability, but also reusability. Deeply nested controls usually has a control somewhere in its hierarchy that is being reused somewhere with possibly the same or similar spec already defined.
* @Manual – this is a special tag that allows us to define specs that should be manually tested while asking the system to ignore the test in automated test runs.
* @OTF11533 – this is the ticket associated with the spec. There can be multiple tickets associated with a spec. Ticket tags are only relevant during release development and can optionally be deleted after the ticket is deployed to production. Unless there are an overwhelming amount of ticket tags on a spec they should be kept for future reference.

With tags in place, we can configure a test run to only test a specific ticket, or page, or any combination of tags. This provides a significant decrease in the amount of time it takes to validate new features and bug fixes.

## Writing Specs

* SpecFlow Scenario
  + This is where the metal hits the road and we write tests in C# with the unit test framework backing, selenium driving browser through page model, and SpecFlow providing structure.

## Running Tests

* Local IDE Test
* Local Command Line Test
* Remote Automated Test

## Test Management

* KISS it, then KISS it again
* Managing Environments
* Managing Test Data
* Types of Tests (Feature, Smoke, Functional, Regression)
  + Tickets must pass a progressive battery of tests. Ranging from test that focus on a particular feature to test that execute every active test that we have written.
* Managing Test Suites
* Managing Test Execution
* Test Policy

## Test Reporting

* What to measure?
  + Defect % (how much rework had to be done)
  + Ticket/Sprint/Release On Time % (how efficient is the process)
    - Everything has a due date and we measure was it done or not, no explanation necessary. It doesn’t mean that dev is bad estimates may be bad, requirements may change, there may have been a misunderstanding in the spec. Don’t over complicate it. Was it complete on due date? If yes, it was on time. If not, it was late. Try to address the reason for lateness, but don’t make this a witch hunt.
  + Test Status % Pass vs Fail vs Error vs Inconclusive vs Ignored (how healthy)
    - If there are a bunch of ignored test then we are leaving money on the table in the effort expended to create those tests and the value they could provide. Inconclusive means the tests haven’t been written yet. Error there was an unhandled exception. Fail, the assertion failed. Pass, well we all know what this one is.
  + Test performance (is it getting slower or is it just me?)
    - This is speculative on my end because I have yet to fully implement this with enough data that I could draw conclusions on. The thesis is that if we measure test performance we can track when performance issues were introduced or track degradation from one point to another point. Hopefully, the data can help pinpoint performance issues since tests are tied to tickets and tickets to source code changes, database changes, configuration and environment changes. This implies that these relationships exist in the data we are capturing on tests. We could also be implicit in our performance evaluation by purposely introducing stress from a load generator that is capable of simulating large amounts of load on our app by running our tests. We have to stress Session also, so it is important to run in multiple Sessions.
* Execution Report
  + Presentation of the above metrics after each run
  + How to report it in a way that produces value.
* Analytics Report (comparison over time)
* Executive Management Report
  + Reports distilled into business terms.
  + Strive to quantify results in terms of ROI, revenue, cost saving, time saving…business benefits while highlighting the loss, deficit, expense, estimate inadequacy…business risk.
  + Perceived benefits, support for brand and mission
  + Perceived risks, get you reduced or worse fired.
* Capturing, persisting, and analyzing test result data

## Engineer Setup

* Computer Assignment
* Software Installation – need install instructions
  + Visual Studio
    - AnkhSVN
    - CodeMaid
    - NuGet Package Manager
    - SpecFlow
  + SQL Server
  + Tortoise SVN
  + Git for Windows
  + Tortoise Git
  + BeyondCompare
  + NUnit
  + Remote Desktop Manager
* Environment Setup
  + PNE Source
  + PNE Automation Source
  + PNE Database
  + PNE Automation Database
  + Remote Desktop Accounts

## Future Goals

*Some future goals.*

* Automated Deployment
* Provide a shared source Test Framework for other SunGard teams to take advantage of (like an internal Github social coding thing, we want to expose a repository across the SunGard internal network)
* UI Pixel based comparison tests (assert variations in look and feel)
* Performance tests
* Pen tests
* Cross Browser Tests
* Cross Device Tests